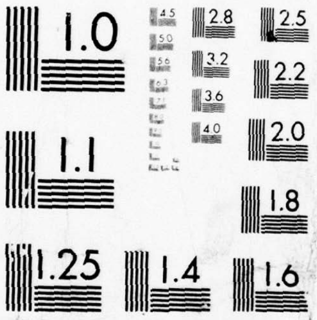
AD-A069 833 ILLINOIS UNIV AT URBANA-CHAMPAIGN APPLIED COMPUTATION--ETC F/G 12/1
A NEW APPROACH TO PLANAR POINT LOCATION.(U)
OCT 78 F P PREPARATA DAAB07-72-C-0259

UNCLASSIFIED ACT-11 NL

1 OF 1
AD
A069833

END
DATE
FILMED
7-79
DDC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

ACT-11 SEPTEMBER 1978

CSL COORDINATED SCIENCE LABORATORY

APPLIED COMPUTATION THEORY GROUP

A NEW APPROACH TO PLANAR POINT LOCATION

FRANCO P. PREPARATA

REPORT R-829 UILU-ENG 78-2222

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) A NEW APPROACH TO PLANAR POINT LOCATION		5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report
		6. PERFORMING ORG. REPORT NUMBER R-829 (ACT-11);UILU 78-2222
7. AUTHOR(s) (10) Franco P. Preparata		8. CONTRACT OR GRANT NUMBER(s) MSC76-17321 DAAB-07-72-C-0259 MCS
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12) 29p.
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		12. REPORT DATE (11) Oct[redacted]978
		13. NUMBER OF PAGES 24
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (14) ACT-11, UILU-ENG-78-2222		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

(15) DAABØ7-72-C-Ø259, NSF-MSC76-17321

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computational Geometry
Analysis of Algorithms
Point Location
Planar Graphs

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Given a planar straight line graph G with n vertices and a point P_0, locating P_0 means to find the region of the planar subdivision induced by G which contains P_0. Recently, Lipton and Tarjan presented a brillian but extremely complex point location algorithm which runs in time $O(\log n)$ on a data structure using O(n) storage. This paper presents a practical algorithm which runs in less than $6 \lceil \log_2 n \rceil$ comparisons on a data structure which uses $O(n \log n)$ storage, in the worst case. The method rests crucially on a simple partition → next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

JOB

410 809

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

20. Abstract (continued)

of each edge of G into O(logn) segments.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

UILU-ENG 78-2222

A NEW APPROACH TO PLANAR POINT LOCATION

by

Franco P. Preparata

This work was supported in part by the National Science Foundation under Contract MSC76-17321 and in part by the Joint Services Electronics Program (U.S. ARmy, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

A NEW APPROACH TO PLANAR POINT LOCATION[†]

Franco P. Preparata
Coordinated Science Laboratory[*]
University of Illinois at Urbana

September, 1978

Abstract

Given a planar straight line graph G with n vertices and a point P_0, locating P_0 means to find the region of the planar subdivision induced by G which contains P_0. Recently, Lipton and Tarjan presented a brilliant but extremely complex point location algorithm which runs in time O(logn) on a data structure using O(n) storage. This paper presents a practical algorithm which runs in less than $6\lceil \log_2 n \rceil$ comparisons on a data structure which uses O(nlogn) storage, in the worst case. The method rests crucially on a simple partition of each edge of G into O(logn) segments.

[†]This work was supported in part by the National Science Foundation under Grant MCS76-17321 and in part by the Joint Services Electronics Program under Grant DAAB-07-72-C-0259.

[*]Also Departments of Electrical Engineering and of Computer Science

1. Introduction

The problem of locating a point in a planar subdivision - briefly called "point location" - is quite important in computational geometry and has received considerable attention in the recent past. It is stated as follows: Given a connected planar straight-line graph G on n vertices and a point P_0, find which region of the planar subdivision induced by G contains P_0.

An early solution to this problem was proposed by Dobkin and Lipton [1], whose location algorithm runs in time O(logn) on a data structure which uses $O(n^2)$ space and can be built in $O(n^2)$ time. More recently Lee and Preparata [2] [3] developed an $O(\log^2 n)$(1) time location algorithm on a data structure constructed in O(nlogn) time and using O(n) space. Observing the trade-off between space/preprocessing on one side and search time on the other, Shamos [4] raised the question of whether O(logn) search time was achievable with less than quadratic storage. This issue was definitively settled by Lipton and Tarjan [5] who showed that the point location problem - called by them "triangle problem" - could be solved in O(logn) time on a data structure which uses O(n) space and can be constructed in time O(nlogn). Their brilliant method, which is based on a theoretically far-reaching planar separator theorem [6], is, however, algorithmically extremely complicated; to quote Lipton and Tarjan themselves, "... this algorithm [is not advocated] as a practical one, but its existence suggests that there may be a practical algorithm with O(logn) time bound and O(n) space bound".

(1)All logarithms in this paper are to the base 2.

The result presented in this paper comes very close to providing a complete substantiation of the above conjecture; specifically, we shall exhibit a practical point location algorithm which runs in O(logn) time on a data structure, which can be constructed in O(nlogn) time, but which uses O(nlogn) space rather than just O(n).

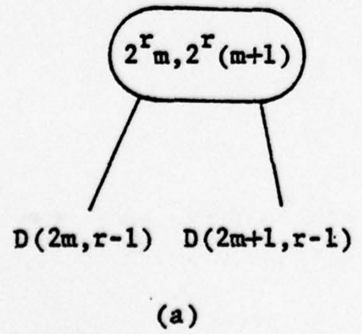
Our method could be viewed as an evolution of the original technique of Dobkin and Lipton [1], which we now briefly review. A horizontal line is drawn through each vertex of G, thereby slicing the plane into horizontal strips called "slabs"; each slab contains no vertex of G and is subdivided by the transversal edges into an ordered set of O(n) regions. Point location is accomplished by first searching the horizontal lines to locate a slab and by next searching the segments crossing the slab to locate a region. Clearly this search is carried out in O(logn) comparisons, but since an edge is partitioned by O(n) horizontal lines, $O(n^2)$ storage is used. In contrast, our method interleaves tests against horizontal lines and test against edges; thus it will not be necessary to decompose the edges in O(n) portions. In particular, the method rests crucially on the observation that each edge of G can be decomposed uniquely into O(logn) segments.

2. Logarithmic segmentation of edges

Let a point v in the plane (x,y) be given as a pair of coordinates x(v) and y(v) and let $\{v_0,\ldots,v_{n-1}\}$ be the vertex set of G, where the numbering is such that $y(v_0) \leq y(v_1) \leq \ldots \leq y(v_{n-1})$. (In the sequel we shall assume for simplicity that these ordinates are distinct; the details of the general case are straightforward.) For additional simplification and without loss of generality we may assume that $y(v_i) = i$; so, when we say that the ordinate of a point u is i we mean $y(u) = y(v_i)$.

Each edge is to be partitioned into a collection of segments; each of these segments will be simply denoted by the ordered pair of ordinates of its extremes. The set of pairs of ordinates delimiting segments is $S = \{(2^k j, 2^k(j+1)) \mid j,k \text{ nonnegative integers}\}$. We want to partition each edge into a minimal number of such segments: for example, edge (9,21) will be partitioned into (9,10)(10,12)(12,16)(16,20)(20,21).

For any given pair of nonnegative integers m and r we define the set S(m,r) - a subset of S - as follows:
$S(m,r) \overset{\Delta}{=} \{(2^k j, 2^k(j+1)) \mid 2^r m \leq 2^k j, 2^k(j+1) \leq 2^r(m+1)\}$. The elements of S(m,r) are organized as the nodes of a full binary tree D(m,r) as shown in figure 1a (a similar structure has been called range tree by Bentley [11]). In figure 1b we show the complete tree D(0,3).



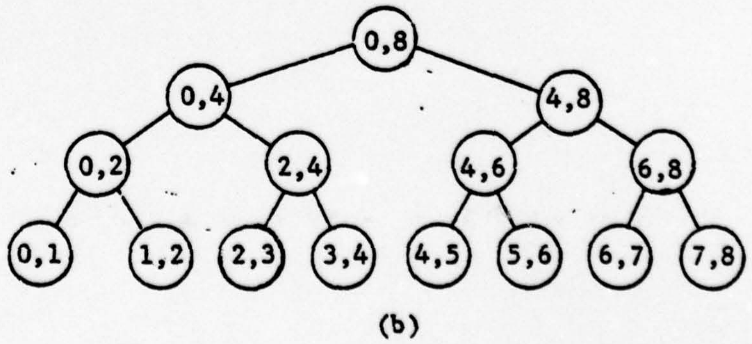


Figure 1. Definition of tree D(m,r) and illustration of D(0,3).

Given an edge (v_i,v_j), with $0 \leq i < j \leq n-1$, we can now produce the desired logarithmic segmentation of it with the aid of the tree $D(0,\lceil \log(n-1) \rceil)$, simply referred to as D. This is accomplished by the following algorithm SEGM((i,j),D) (here $\mathcal{L}_0, \mathcal{L}_1$, and $\mathcal{L}_2$ are lists, and "$\circ$" denotes list concatenation). The segmentation is performed on the vertical projection (i,j) of (v_i,v_j).

```
                          SEGM((i,j),D)
begin   V ← ROOT(D)
        (p,q) ← (L(V),R(V))
        If (i,j) = (p,q) then $\mathcal{L}$ ← (p,q)
        else If j ≤ (p+q)/2 then $\mathcal{L}$ ← SEGM((i,j), LEFTSUBTREE(V))
             else If i ≥ (p+q)/2 then $\mathcal{L}$ ← SEGM((i,j),RIGHTSUBTREE(V))
                  else begin $\mathcal{L}_1$ ← SEGM((i,(p+q)/2), LEFTSUBTREE(V))
                             $\mathcal{L}_2$ ← SEGM(((p+q)/2,j), RIGHTSUBTREE(V))
                             $\mathcal{L}$ ← $\mathcal{L}_1 \circ \mathcal{L}_2$
                       end
        return $\mathcal{L}$
end
```

For example, SEGM((1,7), D) produces $\mathcal{L}$ = (1,2),(2,4),(4,6),(6,7). The action of SEGM can be viewed as tracing two paths - possibly with a common initial subpath - from the root of D to two of its leaves. The number of recursive calls is therefore at most twice the depth of D, and since each new call takes time bounded by a constant, SEGM runs in time O(logn).

We now state without proof properties which follow directly from the algorithm SEGM:

Proposition 1.

An edge (i,j) with $0 \leq i < j \leq n-1$ is partitioned by SEGM in at most $2\lceil \log(n-1)\rceil - 2$ segments;

Proposition 2.

Let (h,k) be a segment, with $h < k$ (h and k are the ordinates of the two extremes of the segment). If $h = 2^r \cdot h'$ with h' odd, then $k \in \{2^r \cdot h' + 2^i | \quad i=0,1,\dots,r.\}$.

3. Construction of the point location tree

We shall now construct the data structure $\mathcal{T}$ - a binary search tree - to be used by the point location algorithm.

Without loss of generality we may assume that the given planar straight-line graph G with n vertices be a triangulation; if not, G can be transformed into one in time O(nlogn) by adding edges according to the algorithm of Garey et al. [7].

The graph G is also assumed to be given as a collection of ordered edge lists; specifically, we let $\mathcal{E}_j = \{(j,i) | (j,i)$ is an edge of G and $i > j\}$ and we assume that the members of $\mathcal{E}_j$ are ordered clockwise around v_j. This representation is obtainable in time O(nlogn) from the more conventional representation $\{$Edges incident on $v_j | j = 0,\dots,n-1\}$.

A preliminary task is the logarithmic segmentation of the edges of G. Let G have m edges. Each edge e of G is partitioned into a string of segments by means of the algorithm SEGM outlined in the previous section. With each segment t we associate an integer, height(t), which is the ordinate of its upper extreme. The string of segments of e is stored as an ordered list $(t_1, t_2, \ldots, t_r)$, where the order is such that $\text{height}(t_1) < \text{height}(t_2) < \ldots < \text{height}(t_r)$: t_1 and t_r are respectively the initial and terminal segments in the list. Since each edge can be partitioned in time O(logn) and $m \le 3n-6$, the entire edge segmentation task runs in time O(nlogn) and uses O(nlogn) space.

The procedure which constructs the data structure $\mathcal{T}$ is called ORGANIZE and has access both to the set of lists $\{\mathcal{E}_j \mid j = 0,\ldots,n-2\}$ and to the m ordered lists of segments. Specifically, it starts with the initial segments of the edges issuing from v_0, processes them and proceeds by acquiring the "upward continuation(s)" of each of the processed segments. This is easily done as follows, where we assume that segment t is contained in some list σ. We also denote by L(t) a list of segments which are the upward continuations of t; L(t) is referred to by a pointer b(t) associated with t.:

```
If t is terminal in $\sigma$ (Comment: t reaches vertex $v_{\text{height}(t)}$ of G) then
    begin  $\mathcal{E} \leftarrow \mathcal{E}_{\text{height}(t)}$
           $\mathcal{E}_{\text{height}(t)} \leftarrow \Lambda$(2)
           If $\mathcal{E} \neq \Lambda$ then L(t) $\leftarrow$ string of initial segments of edges in $\mathcal{E}$
           else L(t) $\leftarrow \Lambda$
    end
else L(t) $\leftarrow$ successor of t in $\sigma$
```

(2) Here and hereafter, Λ denotes the empty sequence.

The procedure also makes use of two auxiliary functions, JOIN and BALANCE, to be discussed in detail later: presently, suffice it to say that JOIN joins together two binary trees by providing a common root, while BALANCE structures a forest of binary trees into a single binary tree.

We shall now informally describe, and illustrate with an example, the procedure ORGANIZE $(\mathcal{L},h,k)$, where h and k are integers $(h \leq k)$ and:

(i) $\mathcal{L}$ is a string of segments, which have the properties that the ordinates of their lower extremes are identical and equal to h, while the ordinates of their upper extremes are no greater than k (descriptively we say that the segments in $\mathcal{L}$ are contained in the <u>horizontal slab</u> $[h,k]$).

(ii) Either $k = n-1$ or, letting $h = h' \cdot 2^r$ (odd h'), $k = h + 2^{\ell}$ for some $0 \leq \ell \leq r$.

Notationally, for some terms $a_1,\ldots,a_r$, $(a_1,\ldots,a_r)$ denotes a string, while $\langle a_1,\ldots,a_r \rangle$ denotes a binary tree so that the string $(a_1\ldots,a_r)$ is obtained when the tree is traversed in inorder ([8],p.). If A_1 and A_2 are two strings, (A_1,A_2) is their concatenation. The procedure also makes convenient use of stacks $\mathcal{L}$,S,U, $\mathcal{B}$; following [8], for a stack S, "$x \Leftarrow S$" denotes that x is the element which has been "popped", while "$S \Leftarrow x$" denotes that x has been "pushed" into the stack. When a string is stored in a stack, its left-most term is at the top of the stack.

```
                    ORGANIZE ($\mathcal{L}$,h,k)(see Comment 1, below)
1.  begin  S ← $\Lambda$, j ← k, $j_0$ ← 0
2.          If h $\neq$ k then  (see Comment 2)
3.            While  $j_0 < k$ do (see Comment 2)
4.                 begin  While $\mathcal{L} \neq \Lambda$ and height(TOP($\mathcal{L}$)) $\leq$ j do
5.                           begin  x $\Leftarrow$ $\mathcal{L}$
6.                                  j ← height(x),
7.                                  Form L(x) and b(x) (see Comment 3)
8.                                  S $\Leftarrow$ (x,b(x))
                             end  (see Comment 4)
9.                       While height(TOP(S)) = j do
10.                          begin (x,b(x)) $\Leftarrow$ S
11.                                 U $\Leftarrow$ x
12.                                 $\mathcal{B}$ $\Leftarrow$ L(x)
                             end (see Comment 5)
13.                      $a_1$ ← BALANCE(U) (see Comment 6)
14.                      ($a_2$,b($a_2$)) ← ORGANIZE ($\mathcal{B}$,j,min(2j-h,k)) (see Comment  )
15.                      If $a_2 \neq \Lambda$ then a ← JOIN ($a_1$,$a_2$),b(a) ← b($a_2$)
                         else a ← $a_1$, L(a) ← $\mathcal{B}$ (see Comment 8)
16.                      S $\Leftarrow$ (a,b(a))
17.                      $j_0$ ← j
18.                      j ← min (2j-h,k) (see Comment 9)
                 end
19.         return S
    end
```

Comment 1. [Referring to the graph in Figure 2, we consider ORGANIZE($\mathcal{L}$,4,8); segments are indicated by means of integers and $\mathcal{L}$ = (15,16, 17,18,19,20,21,22,35); TOP($\mathcal{L}$) is segment 15.]

Comment 2. The major controls of the algorithm are embodied by Steps 2 and 3. Obviously, if h = k, the horizontal slab is empty and the empty tree is returned (Steps 2 and 19). Moreover - as we shall see (Comments 8 and 9) - processing is completed when the control variable j_0 becomes equal to k.

Comment 3. The string L(x) of the upward continuations of segment x is constructed and referred to via b(x), as previously outlined.

Comment 4. Loop 4-8 finds the longest prefix $\mathcal{L}^*$ of string $\mathcal{L}$ so that the terms of (S,$\mathcal{L}^*$) have nonincreasing heights; $\mathcal{L}^*$ is removed from $\mathcal{L}$ and concatenated with S. Specifically two-field records (x,b(x)) are entered into S. [In our example, S becomes (19,18,17,16,15).]

Comment 5. Loop 9-12 finds the longest suffix of S of elements with constant heights, removes it from S and places it into a stack U. Also for each x transferred from S to U the list L(x) (pointed to by b(x)) of the upward continuations of x is placed into a stack $\mathcal{B}$. [In our example, at this point we have S = (16,15), U = (17,18,19) and $\mathcal{B}$ = (26,25,24).]

Comment 6. The function BALANCE - to be described in Section 4 - structures the terms of U into a binary tree to be denoted as HT_1(U). Each of these terms is stored as a node in the search data structure $\mathcal{T}$; a_1 refers to the node storing the root of HT_1(U), and is itself treated as a "term".

Comment 7. This recursive call obtains a tree $HT_2(U)$, again referred to through its root in a_2; $b(a_2)$ points to the string of segments which are the upward continuations of $HT_2(U)$. [In our example, $HT_2(U) = \langle 24,25,26 \rangle$ and $b(a_2)$ points to the string (28,29,30).]

Comment 8. If both $HT_1(U)$ and $HT_2(U)$ are nonempty, they are joined together into a new binary tree VT(U) - referred to via its root a - and the upward continuations of $HT_2(U)$ become the upward continuations of VT(U) [in our example, $VT(U) = \langle\langle 17,18,19 \rangle * \langle 24,25,26 \rangle\rangle$]; otherwise, when $HT_2(U)$ is empty, the string β itself gives the upward continuations. Notice that $HT_2(U)$ is empty only when in Step 14 we have a call ORGANIZE(β,k,k), i.e., when j = k.

Comment 9. A new "term" (a,b(a)) is formed in Step 16 and pushed into S [in our example, S becomes $(\langle\langle 17,18,19 \rangle * \langle 24,25,26 \rangle\rangle,16,15)$]. Notice that the major loop 4-18 is repeated until j_0 is set equal to k in Step 17, which occurs exactly just after Step 14 returns the empty tree.

In our example, ORGANIZE ((15,16,17,18,19,20,21,22,35),4,8) produces the tree $a = \langle 15\langle\langle 16,\langle\langle 17,18,19\rangle * \langle 24,25,26\rangle\rangle ,20\rangle * \langle\langle\langle 27,28\rangle * \langle 32,33,34\rangle\rangle ,29,30\rangle\rangle ,21,22,35\rangle$; b(a) points to the string (36,37,38,39,40,41).

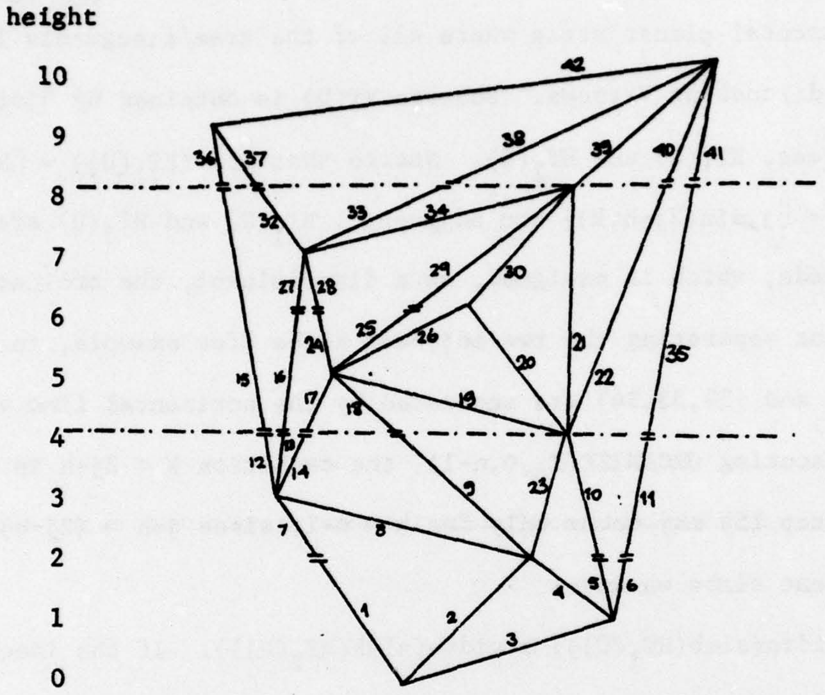


Figure 2. Illustration for the procedure ORGANIZE

Once the function ORGANIZE is available, denoting by $\mathcal{L}_0$ the string of the initial segments of the edge string $\mathcal{E}_0$, the construction of the search structure $\mathcal{T}$ for graph G is trivially done by the single call ORGANIZE($\mathcal{L}_0$,0,n-1). In our example, $\mathcal{L}_0$ = (1,2,3).

The construction of subtrees occurs in Steps 14, 15, and 16. As we noted there are two types of subtrees - H-trees and V-trees -, depending upon the way they are generated. The root of a V-tree is said to be a V-node, while all others are referred to as H-nodes.

A subtree is <u>said to contain a segment t</u> if t has been assigned to one of its nodes; a subtree is <u>said to contain a vertex v of G</u> if <u>v is in the interior</u> of the trapezoid which is the convex span of the points of the segments contained in the subtree; for each subtree we define its <u>slab</u> as the smallest horizontal planar strip where all of the tree's segments lie.

We begin by discussing V-trees. Subtree VT(U) is obtained by "joining" together two H-trees, $HT_1(U)$ and $HT_2(U)$. Notice that $slab(HT_1(U)) = [h,j]$ and $slab(HT_2(U)) = [j,min(2j-h,k)]$ are adjacent; $HT_1(U)$ and $HT_2(U)$ are joined by means of a V-node, which is assigned, as a discriminant, the ordinate of the horizontal line separating the two adjacent slabs (for example, in Figure 2, $\langle 27,28\rangle$ and $\langle 32,33,34\rangle$ are separated by the horizontal line $y = 7$). Notice that in executing ORGANIZE($\mathcal{L}_0$,0,n-1), the condition $k < 2j-h$ in a recursive call (Step 15) may occur only for $k = n-1$; since $j-h = (2j-h)-j$, for the two adjacent slabs we have:

<u>Proposition 3</u>. $width(slab(HT_1(U))) \geq width(slab(HT_2(U)))$. If the inequality is strict, then $slab(HT_2(U))$ is upper bounded by the line $y = n-1$.

We define the <u>level</u> of VT(U) as $\log \max_{i=1,2} (width(slab\ HT_i(U))) + 1$. Since a slab of width w (an integer) contains exactly w-1 vertices of G in its interior, we obviously have that a V-tree of level i contains at most 2^i-1 vertices of G. On the other hand any V-tree - except possibly one whose slab is upper bounded by $y = n-1$ - contains at least one vertex of G. Thus we have:

<u>Proposition 4</u>. All V-trees - except possibly one - contain at least one vertex of G; if <u>level</u> (T) = i, then T contains at most 2^i-1 vertices of G.

The number of V-nodes is obtained as follows. Let $\mathcal{P}=\{T|T$ is a V-tree and there is no other V-tree T' which is a proper subtree of T$\}$. The root of any $T \in \mathcal{P}$ is the only V-node in T, otherwise there would be V-tree T' which is a subtree of T. The cardinality of $\mathcal{P}$ is at most (n-2), since v_0 and v_n are not contained in any V-tree. For each $T \in \mathcal{P}$, suppose to trace the path from its root V to the root of $\mathcal{T}$ and let $V_2, V_3, \ldots, V_p$ be the sequence of the V-nodes encountered; V_j is the root of some V-tree T_j, and obviously level (T) < level (T_1) < ... < level (T_p). Since the level of any V-tree is upper-bounded by $\lceil \log n \rceil$, and $|\mathcal{P}| \leq n-2$, we conclude that the number of V-nodes is upper-bounded by O(nlogn).

We now consider the other type of subtrees, the H-trees. They are formed by structuring (Step 14) into a binary tree a mixed sequence U of segments and V-trees, all spanning the same horizontal slab (e.g., 16, ⟨⟨17,18,19⟩*⟨24,25,26⟩⟩, and 20 in Figure 2). In general U is the form $\tau_0 T_1 \tau_1 T_2 \ldots \tau_{r-1} T_r \tau_r$, where the T_j's are V-trees of identical level and $\tau_0, \ldots, \tau_r$ are each a string of segments; we claim that none of the strings $\tau_1, \ldots, \tau_{r-1}$ is empty. To prove this, notice that each T_j is the join of two H-trees H_{1j} and H_{2j}; if τ_i is empty, for $1 \leq i \leq r-1$, the procedure ORGANIZE would combine the members of H_{1i} and $H_{1,i+1}$ into a single tree, before examining the members of H_{2i} and $H_{2,i+1}$, thus contradicting the existence of T_i and T_{i+1}. The nodes created in structuring U are H-nodes and to each one of them we assign one of the segments in $\tau_0 \cup \tau_1 \cup \ldots \cup \tau_r$, and a (discriminant) linear function f(x,y), so that f(x,y) = 0 is the equation of the line containing that segment. The details of the construction of $HT_1(U)$ and $HT_2(U)$ - by the subroutine BALANCE - will be discussed in the next section in connection with the performance analysis of the method; presently, we just note that the number of V-nodes (i.e. of V-trees) involved in the structuring process is just O(n), rather than O(nlogn).

To see this, we reduce the tree $\mathcal{T}$ to a tree $\mathcal{T}_V$ which contains only V-nodes and is constructed as follows: delete and bypass all the H-nodes of $\mathcal{T}$ one at a time, i.e., for each non-leaf H-node V replace the three arcs (FATHER(V),V), (V,LEFTSON(V)), and (V,RIGHTSON(V)) with the two arcs (FATHER(V),LEFTSON(V)) and (FATHER(V),RIGHTSON(V)); a leaf H-node is just suppressed. Clearly $\mathcal{T}_V$ has at most (n-2) leaves. The nodes of $\mathcal{T}_V$ are of three types: the <u>regular</u> ones with two or more "children", the <u>singular</u> ones with exactly one child, and the <u>leaves</u>; it is clear that only the children of regular nodes take part in the balancing process. Therefore suppose now to further delete and bypass every singular node; the resulting tree is such that its non-leaf nodes have at least two children and, since there are at most (n-2) leaves, there are at most 2n-5 nodes altogether. This proves the claim.

If we represent H-nodes by the symbol $\textcircled{t}$, where t is the number of the segment assigned to the node, and V-nodes by the symbol $\overset{y}{\triangledown}$, where y is the ordinate assigned to the node, the structure $\mathcal{T}$ for the graph of Figure 2 is shown in Figure 3.

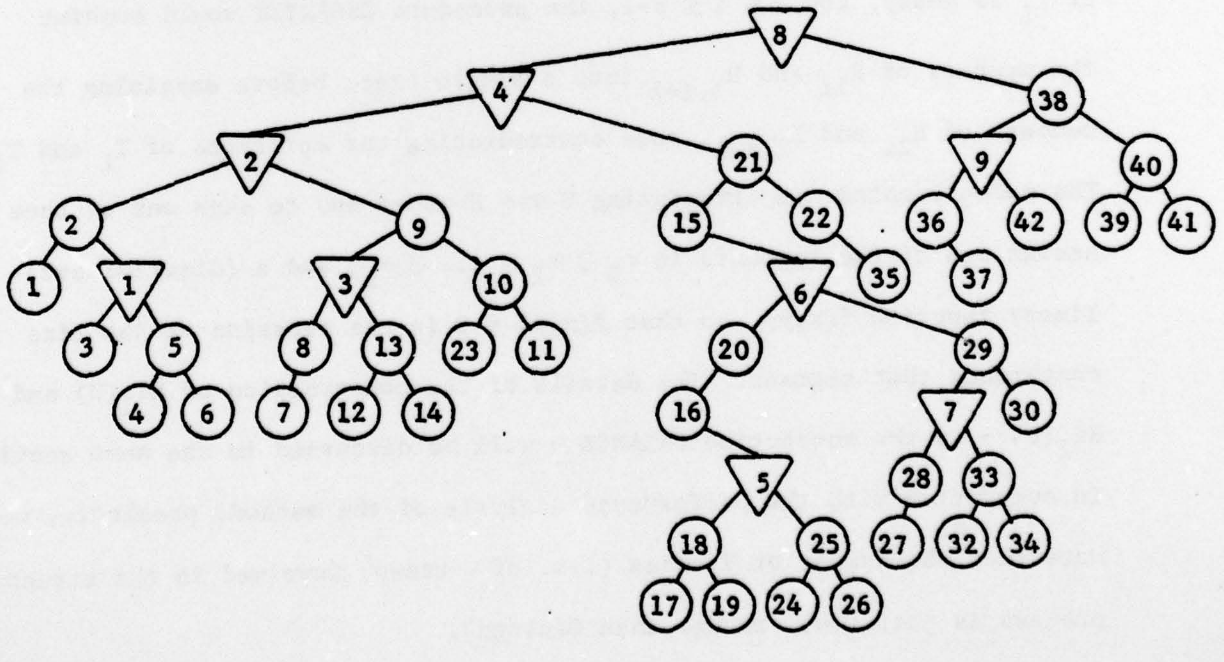


Figure 3. The binary search tree $\mathcal{T}$ for the graph of Figure 2.

4. Performance analysis of the method

We begin by evaluating the depth of the search tree $\mathcal{T}$. Clearly $\mathcal{T}$ has O(nlogn) nodes. In fact the H-nodes are in a one-to-one correspondence with the set of segments, and the latter has cardinality O(nlogn); as to the V-nodes, we have just shown that their number is also O(nlogn).

If $\mathcal{T}$ were balanced, it would have depth O(logn). However there is no explicit provision in the ORGANIZE algorithm to achieve such property; as a matter of fact, the depth of $\mathcal{T}$ critically depends on the subroutine BALANCE used for structuring H-trees. Indeed, suppose that in Step 14 of ORGANIZE, the set U contains O(n) V-trees. The increase in depth produced by BALANCE(U) could be O(logn), thereby resulting in an $O(\log^2 n)$ depth for $\mathcal{T}$. However, we shall now describe a procedure BALANCE which produces a global O(logn) depth for $\mathcal{T}$. The procedure is based on the following lemmas (the first of which is a variant of another lemma presented in [9]):

Lemma 1. Let $\mathcal{a} = a_1 a_2 \ldots a_p$ be a string with $p > 1$ and let the positive integer $|a_j|$ denote the weight of a_j; also, let $|\mathcal{a}| = \Sigma_{j}^{p} {}_{1} |a_j|$ and $M = \max_{j=1}^{p} |a_j|$. Then for any number $M \leq m < |\mathcal{a}|$, the string $\mathcal{a}$ can be algorithmically partitioned as $\mathcal{a} = \mathcal{a}_1\mathcal{a}_2\mathcal{a}_3\mathcal{a}_4$ so that $|\mathcal{a}_2| \leq m$, $|\mathcal{a}|_3 \leq m$, and $|\mathcal{a}_2| + |\mathcal{a}_3| > m$.

Proof: Arrange the terms of $\mathcal{a}$ as the leaves of a balanced binary tree $t(\mathcal{a})$ and for each node V of this tree $t(\mathcal{a})$ compute the weight $|V|$ as $|\text{LEFTSON}(V)| + |\text{RIGHTSON}(V)|$; obviously $|\text{ROOT}(t(\mathcal{a}))| = |\mathcal{a}|$. If we trace a path from the root of $t(\mathcal{a})$ following at each node the branch of larger weight, the weights of the traversed nodes form a decreasing sequence whose minimum is guaranteed to be no larger than M. Thus there is a unique node V^* on this path such that $|V^*| > m$, $|\text{LEFTSON}(V^*)| \leq m$, $|\text{RIGHTSON}(V^*)| \leq m$. We then let $\mathcal{a}_2 :=$ string of leaves of LEFTSON(V^*), $\mathcal{a}_3 :=$ string of leaves of RIGHTSON(V^*), while $\mathcal{a}_1$ and $\mathcal{a}_4$ are the (possibly empty) prefix and suffix of $\mathcal{a}$. □

For a V-tree T we define its weight $|T|$ as the number of vertices of G contained in T (recall that, by Proposition 4, $(\text{level}(T) = 1) \Rightarrow (|T| = 1)$). An *ℓ-string* has the form $U = \tau_0 T_1 \tau_1 \dots \tau_{r-1} T_r \tau_r$, where the T_j's are V-trees of identical level $\ell \geq 1$ and the τ_i's are (possibly empty) strings of segments. We define the *weight* $|U|$ of U as $\sum_{j=1}^{r} |T_j|$.

Lemma 2. Let $VT = \text{JOIN}(HT_1, HT_2)$. The trees HT_1 and HT_2 can be algorithmically constructed so that $\text{depth}(VT) < \lceil \log n \rceil + 2\log|VT| + 3\text{level}(VT) - 1$.

Proof: For simplicity, let $\delta(T) \overset{\Delta}{=} \text{depth}(T) - \lceil \log n \rceil$. We make the following inductive hypotheses:

P1. If U is a j-string with $0 < |U| < K_1 < K$, then $\delta(U) < 2\log|U| + 3j + 1$;

P2. If T is a V-tree, with $|T| < K$ and $\text{level}(T) = j < \ell$, then $\delta(T) < 2\log|T| + 3j - 1$.

The induction can be started with $j = 1$. In fact $\text{level}(T) = 1$ implies $|T| = 1$, i.e., if $T = \text{JOIN}(H_1, H_2)$, H_1 and H_2 are each trees of $O(n)$ segments, so that $\delta(H_i) \leq 0$ $(i = 1,2)$ and $\delta(T) \leq 1 < 2$. Also, if U is a 1-string, its corresponding slab has width 2. It follows that $|U| \leq 1$ and either $U = \tau' T \tau''$ or $U = \tau$, where τ', τ'', τ are strings of segments and $\text{level}(T) = 1$; in either case $\delta(U) < 4$.

Proof of P1. Let $U = \tau_0 T_1 \tau_1 \dots \tau_{r-1} T_r \tau_r$ and $|U| = K_1$. Notice that $\text{depth}(\tau_i) \leq \lceil \log n \rceil$ (i.e. $\delta(\tau_i) \leq 0$, for every $0 \leq i \leq r$) and let $|T_s| = \max_{j=1}^{r} |T_j|$.

(1) $|T_s| \geq K_1/2$. We express U as $U_1 t_1 T_s t_2 U_2$, where both U_1 and U_2 are j-strings (with $|U_1|, |U_2| \leq K_1/2$) and t_1 and t_2 are segments. Since $|T_s| \leq K_1 < K$ and $\text{level}(T_s) = j$, by P2 we have $\delta(T_s) < 2\log|T_s| + 3j - 1$ $\leq 2\log K_1 + 3j - 1$. With regard to $U_i (i = 1,2)$, either $|U_i| = 0$ (in which case

U_i consists of segments and $\delta(U_i) \le 0$) or by P1 $\delta(U_i) < 2\log|U_i| + 3j + 1$ $\le 2\log K_1 + 3j - 1$. Clearly the tree in Figure 4a structures U so that $\delta(U) < 2\log K_1 + 3j + 1$.

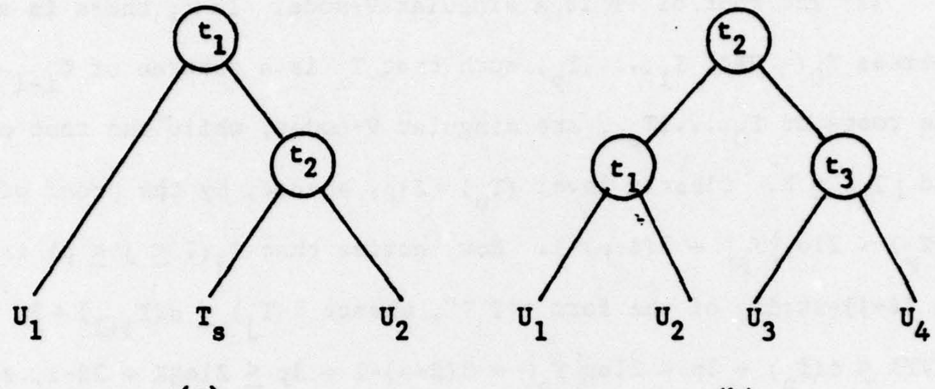


Figure 4.

(2) $|T_s| < K_1/2$. We apply Lemma 1 to the string U with $m=K_1/2$. We obtain the decomposition $U_1t_1U_2t_2U_3t_3U_4$, where t_1,t_2,t_3 are segments and the U_j's are j-strings with $|U_2|,|U_3| \le K_1/2$, and $|U_2| + |U_3| > K_1/2$. The latter implies $|U_1| + |U_4| < K_1/2$, i.e., $|U_1|,|U_4| < K_1/2$. By P1 $\delta(U_1),\delta(U_2),\delta(U_3),\delta(U_4) < 2\log(K_1/2) + 3j + 1 = 2\log K_1 + 3j-1$. Then clearly, the tree in Figure 4(b) structures U so that $\delta(U) < 2\log K_1 + 3j + 1$.

<u>Proof of P2</u>. Let $|VT| = K$ and level(VT) $= \ell$, with VT = JOIN(HT_1,HT_2), $|HT_1| = K_1$, $|HT_2| = K_2$, $(K_1 + K_2 = K)$.

We must now distinguish two cases:

(1) the root of VT is a regular V-node. In this case, $0 < K_1,K_2 < K$. Consider HT_1 (an analogous argument holds for HT_2). The set U is an

$(\ell-1)$-string $\tau_0 T_1 \tau_1 \ldots \tau_{r-1} T_r \tau_r$, with $|U| = K_1 < K$ and level$(U) = \ell-1$. Then, by P1, we have $\delta(U) < 2\log|U| + 3(\ell-1) - 1 = 2\log K_1 + 3\ell - 2$. It follows that $\delta(VT) = \max(\delta(HT_1), \delta(HT_2)) + 1 < 2\max(\log K_1, \log K_2) + 3\ell - 2 + 1 < 2\log K + 3\ell - 1$.

(2) The root of VT is a singular V-node. In $\mathcal{T}$, there is a sequence of V-trees $T_0(= VT)$, $T_1, \ldots, T_p$, such that T_i is a subtree of T_{i-1} (for $i=1,\ldots,p$), the roots of $T_0, \ldots, T_{p-1}$ are singular V-nodes, while the root of T_p is regular, and $|T_i| \leq K$. Clearly level $(T_p) = \ell-p$, whence, by the proof of case (1), $\delta(T_p) < 2\log|T_p| + 3(\ell-p)-1$. Now, notice that $T_j (1 \leq j \leq p)$ is contained in an $(\ell-j)$-string of the form $\tau' T_j \tau''$, whence $\delta(T_j) \leq \delta(T_{j+1}) + 3$. It follows that $\delta(VT) \leq \delta(T_p) + 3p < 2\log|T_p| + 3(\ell-p)-1 + 3p \leq 2\log K + 3\ell-1$, since $|T_p| \leq |VT| = K$. The proof is thus completed. □

In conclusion we have:

<u>Theorem</u>. The depth of the binary tree $\mathcal{T}$ is less than $6\lceil \log n \rceil$.

<u>Proof</u>: If the root of $\mathcal{T}$ is a V-node, then $\mathcal{T}$ is a V-tree of level $\lceil \log n \rceil$ and, by lemma 2, depth $(\mathcal{T}) < \lceil \log n \rceil + \log(n-2) + 3\lceil \log n \rceil - 1 < 6\lceil \log n \rceil - 1$. If the root of $\mathcal{T}$ is an H-node, then there is one edge in G between v_0 and v_{n-1}, and $(n-1)$ is a power of 2. In this case G appears as $G_1 t G_2$ where both G_1 and G_2 are graphs with no more than n vertices; G_1 and G_2 can be structured into binary trees $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, whose roots are V-nodes and heights are less than $6\lceil \log n \rceil - 1$. It follows that the tree structuring G has depth less than $6\lceil \log n \rceil$. □

We shall now estimate the running time of the procedure. First we consider the global work performed by the BALANCE subroutine, described in the proof of Lemma 2. If U contains r V-trees $T_1, T_2, \ldots, T_r$, then, using a result of [10], the balancing runs in time O(rlogr). As we have shown, the total number of V-trees involved in balancing operations is O(n), whence O(nlogn) is the overall running time of BALANCE.

We shall now evaluate the running time of the procedure ORGANIZE. We have just shown that BALANCE uses O(nlogn) steps altogether; analogously JOIN runs in time proportional to the number of V-nodes, i.e., in time O(nlogn). The remaining work is conveniently charged to the individual segments. Specifically, each segment is transferred from $\mathcal{L}$ to S (Steps 5 and 8), and then from S to U (Steps 11 and 12); clearly, the work expended in these transfers is bounded by a constant. When a segment x is transferred from $\mathcal{L}$ to S we associate with it a pointer b(x) (Step 8) to the string L(x) of its upward continuations. The construction of L(x)(Step 7) takes time proportional to its size, so that the global work which is done in Step 7 is proportional to the number O(nlogn) of segments; the construction of b(x) takes constant time. In summary, a segment x is transferred from an original segment list to some list L(t) of "upward continuations" of some other segment t and from here to a stack $\mathcal{B}$; from $\mathcal{B}$ it is next transferred to S and finally to U: clearly the total work involved per segment is bounded by a constant, and since there are O(nlogn) segments, also this portion of the work is O(nlogn). We conclude therefore that the running time of ORGANIZE ($\mathcal{L}_0$,0,n-1) is O(nlogn); that the space used is also O(nlogn) is straightforward.

5. Point location

To locate a point $P_0 = (x_0,y_0)$ in the planar subdivision induced by G, we use $\mathcal{T}$ as a binary search tree. With each H-node of $\mathcal{T}$ which has one or no descendant we append one or two leaves, respectively, and with each such leaf we associate the identifier of a planar region (bordering with the edge associated with the parent H-node). The point location proceeds as follows: at each node V of $\mathcal{T}$ we choose a branch: if V is a V-node, by comparing y_0 with y(V); if V is an H-node, by testing the sign of $f(x_0,y_0)$, where $f(x,y)$ is the discriminant function of V. Thus we trace a unique path from the root to a leaf at which point the point location is completed. By the preceding discussion this process uses a number of comparisons bounded by the depth of $\mathcal{T}$, i.e., $6\lceil \log n \rceil$.

6. Comments and Applications

As the previous analysis indicates, planar point location is simply done in time O(logn) using a search structure which can be stored in O(nlogn) space. Specifically, less than $6\lceil \log n \rceil$ comparisons are ever needed, although the analysis which establishes the upper-bound on the depth of $\mathcal{T}$ is overly generous and a multiplicative constant for $\lceil \log n \rceil$ substantially lower than 6 can be expected.

As to the storage requirement, the analysis refers to the case in which each of O(n) edges is partitioned into O(logn) segments; this intuitively corresponds to a large fraction of long edges, which presumably is not the average case; however, graphs can be constructed for which this situation occurs. It is conceivable that the simple approach presented in this paper could be further refined to achieve O(n) storage while maintaining O(logn) search time.

Notice that the described point location method is not restricted to triangulations, nor to planar subdivisions induced by straight-line graphs. Indeed the straight-line segments may be replaced by other curves if the following two properties hold: (i) the curves are single-valued in one selected coordinate (say, y), and (ii) the discrimination of a point with respect to any of the curves can be done in constant time. For example these conditions are clearly met by arcs of circle or of other conics if they have no horizontal tangent, except possibly at their extremes. We can now mention two applications of the given method. Both problems have recently received consideration in the literature [11,12].

1. Fixed-radius near neighbor searching. This problem involved finding all points of a set F in the plane which are within some fixed radius r of a "query point"[11]. Bentley and Maurer have recently proposed - among other methods - a locus approach, which consists in subdividing the plane into regions each of which is the locus of the points within distance r from a given subset F' of F (this region is clearly the intersection of all the circles with radius r centered at the points in F'). Let $F = \{p_1,\ldots,p_n\}$, and let C_i be the circle of radius r with center in $p_i \in F$. For each C_i, let u_i and ℓ_i be the two points on the circle C_i with largest and smallest ordinates, respectively, and let I denote the set of intersections of pairs of circles in $\{C_i | i=1,\ldots,n\}$. If we define $V \overset{\Delta}{=} I \cup \{u_i | i=1,\ldots,n\} \cup \{\ell_i | i=1,\ldots,n\}$, the circumference of each C_i is partitioned into a set of arcs which have properties (i) and (ii) given above. Therefore V is the vertex set of a planar graph G whose edges are the arcs just described. To this planar graph the method of this paper is applicable. Since $|V| = |I| + |\{h_i | i=1,\ldots,n\}| + |\{\ell_i | i=1,\ldots,n\}| = 2\binom{n}{2} + n + n = n(n+1)$, graph G is planar with $O(n^2)$ vertices. Thus fixed-radius near-neighbor searching can be solved in O(logn) time with a data structure using $O(n^2 \log n)$ space and constructible in $O(n^2 \log n)$ time; in [11] the latter two quantities are both $O(n^3)$.

2. Maxima testing in three dimensions. For points u and v in three-dimensional Euclidean space $\mathbb{R}^3$, u is said to dominate v if $x_i[u] \geq x_i[v]$ $(i=1,2,3)$. Given a finite set F of points in $\mathbb{R}^3$, $u \in F$ is a maximum of F

if it is not dominated by any other point in F. Suppose now that F is a set of maxima of F; testing a target point p for maximum in F means to determine if there is at least a point $u \in F$ which dominates p.

Letting $|F| = n$, Bentley [12] solves this problem in $O(\log^2 n)$ time on a search data structure that is stored in $O(n\log n)$ space and is constructed in $O(n\log n)$ time. We now show that the same storage and preprocessing time can be maintained while reducing the test time to $O(\log n)$.

Let $F = \{u_1, \ldots, u_n\}$. Let v be the point such that $x_j[v] = \min_{i=1}^{n} x_j[u_i]$ $(j=1,2,3)$; for convenience we may assume that v be the origin of $\mathbb{R}^3$, so that all points of F lie in the positive orthant $\mathbb{R}^3_+$. Let M_i be the domain of points of $\mathbb{R}^3_+$ dominated by $u_i \in F$, and let $M = \bigcup_{i=1}^{n} M_i$. Consider now the surface of M and suppose to project it on one of the coordinate planes, say (x_1, x_2). This projection appears as a planar straight-line graph G, each finite region of which is the projection of a portion of the surface of M_i, for some i (Figure 5); it follows that if the (x_1, x_2)-projection of the target point

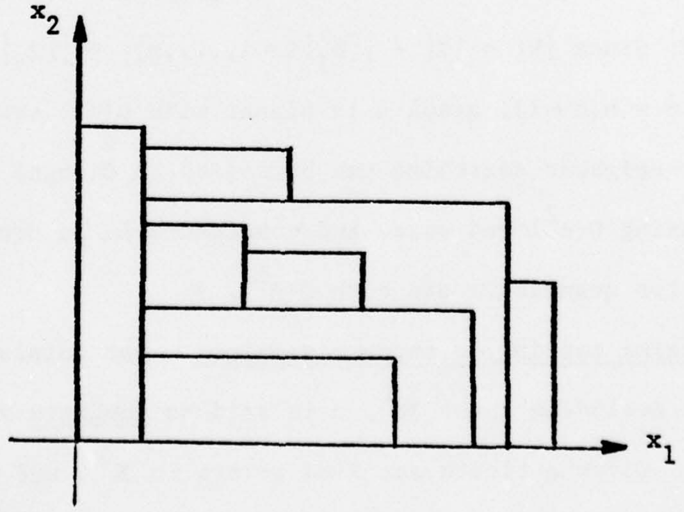


Figure 5. Typical projection of the surface of M on the plane (x_1, x_2). The vertical edges are shown as thick lines.

p falls in the region of G associated with $u_i \in F$, then the maxima testing reduces to comparing $x_3[p]$ with $x_3[u_i]$. Thus maxima testing is done via point-location in G. Notice now that G has two edges - respectively parallel to the x_1 and x_2 axes - issuing from the (x_1,x_2)-projection of each $u_i \in F$. It is easy to realize that the point-location procedure can be applied to the graph consisting of the n edges parallel to, say, the x_2-axis, and the positive x_2-axis itself (see Figure 5). Obviously the search data structure can be stored in O(nlogn) space and is constructible in O(nlogn) time. Referring to the arguments of Bentley [12], the time for worst-case maxima testing in k dimensions can be reduced from $O(\log^{k-1} n)$ to $O(\log^{k-2} n)$ for $k \geq 3$.

References

1. D. P. Dobkin and R. J. Lipton, "Multidimensional searching problems," SIAM Journal on Computing, vol. 5, 181-186 (1976).

2. D. T. Lee and F. P. Preparata, "Location of a point in a planar subdivision and its applications," Proceedings of Eighth ACM Symposium on Theory of Computing (SIGACT), Hershey, Penn., pp. 231-235, May 1976.

3. D. T. Lee and F. P. Preparata, "Location of a point in a planar subdivision and its applications," SIAM Journal on Computing vol. 6, N. 3, pp. 594-606, September 1977.

4. M. I. Shamos, Computational Geometry, Dept. of Comp. Sci., Yale University, 1977. To be published by Springer Verlag.

5. R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," Proceedings of the 18th Symposium on Found. of Computer Science, Providence, R. I., pp. 162-170, October 1977.

6. R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," Conference on Theoretical Computer Science, Waterloo, Ont., pp. 1-10, August 1977.

7. M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, "Triangulating a simple polygon," to appear in Information Processing Letters, Nov. 1977.

8. E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Englewood Cliffs, N. J., 1977.

9. R. P. Brent, D. J. Kuck, and K. Maruyama, "The parallel evaluation of arithmetic expressions without division," IEEE Transactions on Computers, vol. C-22, N. 5, pp. 532-534, May 1973.

10. D. E. Muller and F. P. Preparata, "Restructuring of arithmetic expressions for parallel evaluation," Journal of the ACM, vol. 23, N. 3, pp. 534-543, July 1976.

11. J. L. Bentley and H. A. Maurer, "A note on Euclidean near neighbor searching in the plane, submitted to Information Processing Letters (1978).

12. J. L. Bentley, "Multidimensional divide-and-conquer," Carnegie-Mellon University Research Review, Department of Computer Science, Fall 1977.